第15章 多态性和虚函数

概述

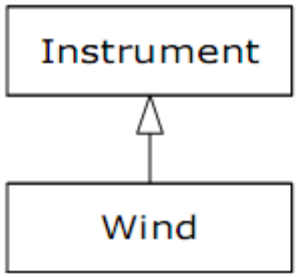
- 多态性(polymorphism)提供了接口与具体实现之间的另一层隔离。从而将”what”与”how”分离。
- 调用*同名函数*却会因为*上下文不同*而有*不同实现*的一种机制。
- 主要内容
  - 多态的使用
  - 多态的内部实现
  - 重载和重定义
  - 多态与构造/析构
  - 多态与向下类型转换

向上类型转换

- 将一个对象作为其基类来使用，称为向上类型转换。
- 然而，仅仅使用继承无法达到期望的向上类型转换的效果。

向上类型转换：继承的问题

```
//: C14:Instrument.cpp
// Inheritance & upcasting
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  void play(note) const {}
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```



是Wind的play还是Instrument的Play被调用了？

向上类型转换：函数调用的捆绑

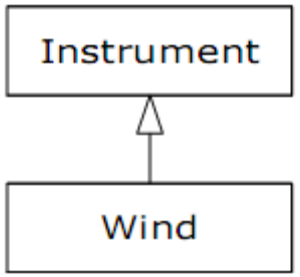
- 将函数体与函数调用相联系称为捆绑(binding)。
- 当捆绑在程序运行之前（由编译器和连接器）完成时，称为早捆绑（early binding）
- 当捆绑在程序运行时完成时，称为晚捆绑（late binding）或运行时捆绑（Runtime binding）。

虚函数：晚捆绑的声明

- 若要一个函数实现晚捆绑，需要用virtual关键字对其进行修饰。
- 仅仅需要在基类中声明一个函数为virtual。所有从该基类的派生类对应的函数都将为virtual。

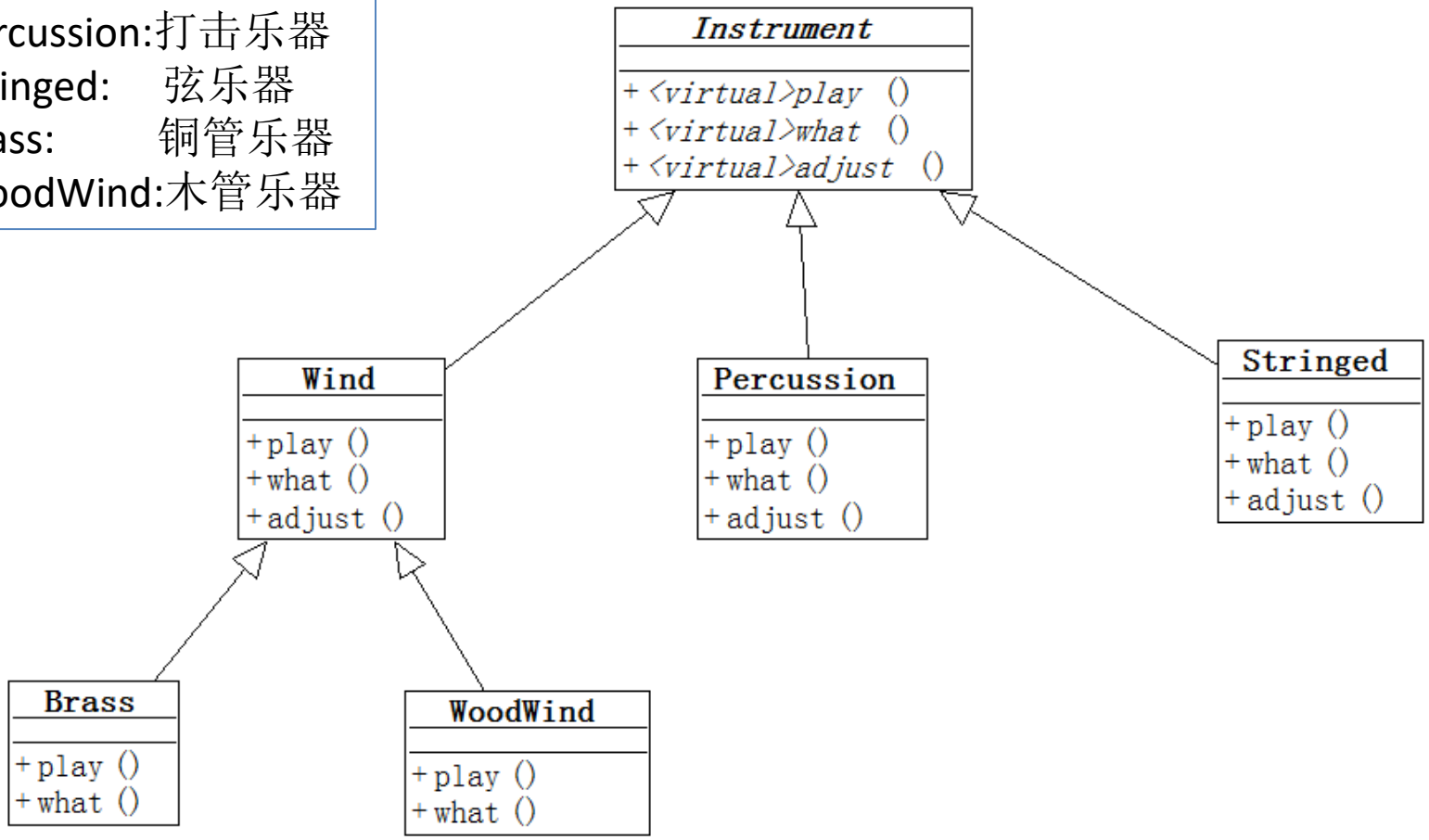
向上类型转换：解决继承问题

```
//: C14:Instrument.cpp
// Inheritance & upcasting
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```



一个更为复杂的例子

Wind: 管号乐器
Percussion:打击乐器
Stringed: 弦乐器
Brass: 铜管乐器
WoodWind:木管乐器



一个更为复杂的例子

```
// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

// Upcasting during array initialization:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};
```

```
int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

类的大小

```
class NoVirtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
};

class OneVirtual {
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};

class TwoVirtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};
```

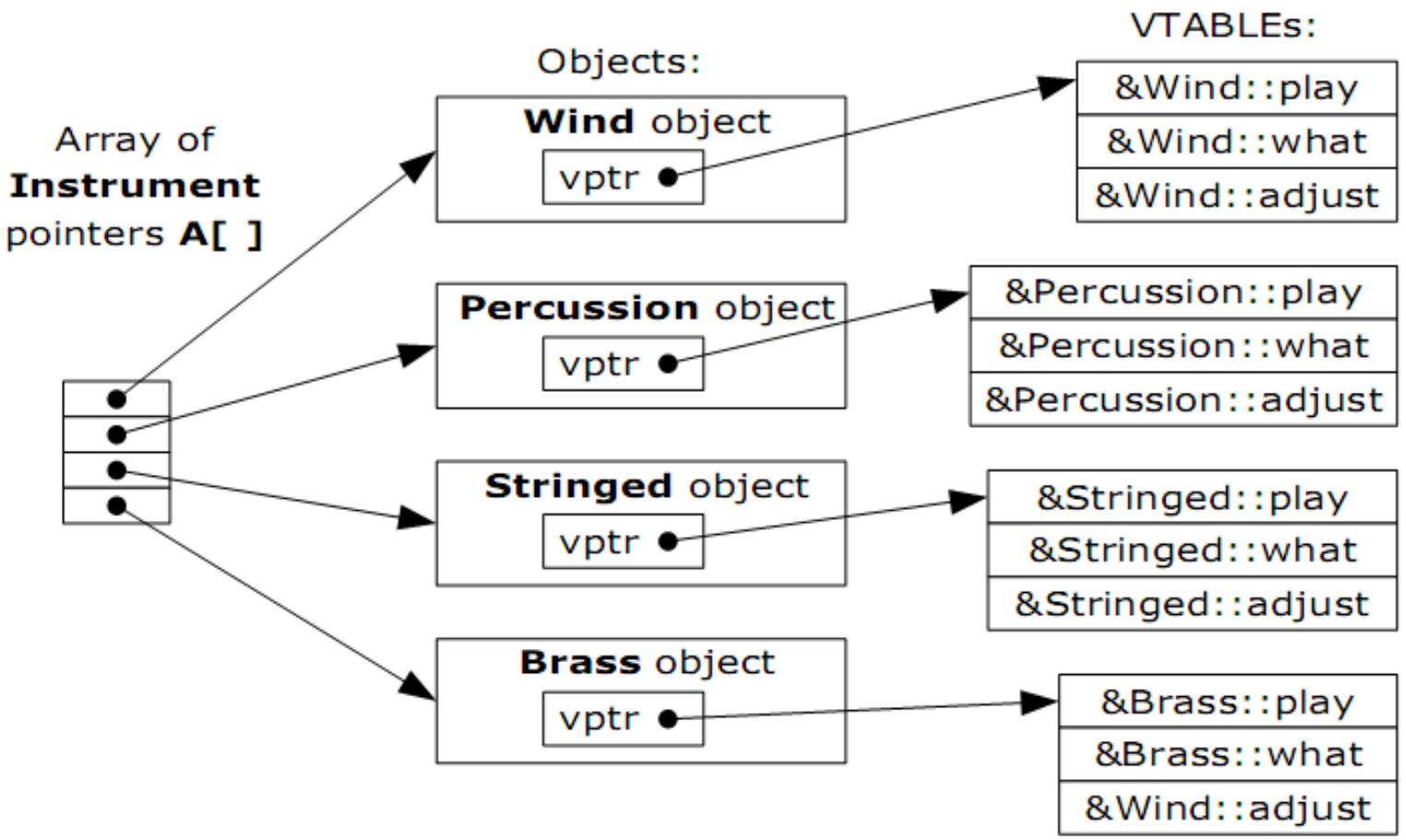
在类实例中存放VTABLE

```
int main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "NoVirtual: "
       << sizeof(NoVirtual) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "OneVirtual: "
       << sizeof(OneVirtual) << endl;
  cout << "TwoVirtuals: "
       << sizeof(TwoVirtuals) << endl;
} ///:~
```

C++如何实现晚捆绑

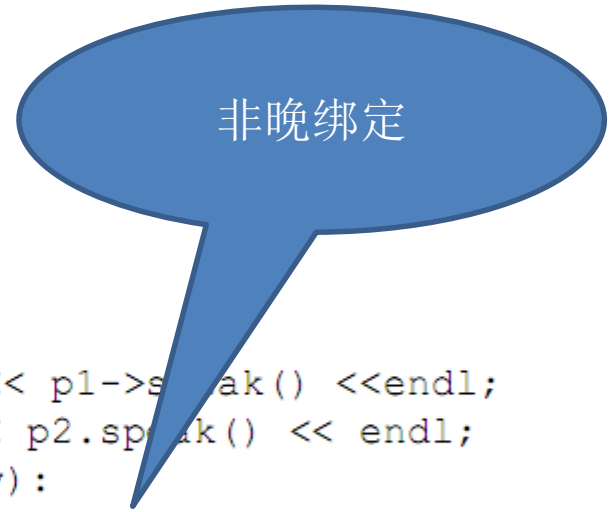
- 一个典型的C程序员通常会很好奇C++的晚捆绑是如何实现的。
  - 而java程序员会认为本来就应该这样，没什么好奇怪的。
- 典型的C++编译器对每个包含虚函数的类创建一个表(称为VTABLE)，该表中保存有特定类的虚函数地址。
- 在每个类的实例中，都有一个指向该地址的指针VPTR。

虚函数功能图示


VTABLEs:
Objects:
Array of
Instrument
pointers A[ ]
Wind object
vptr
&Wind::play
&Wind::what
&Wind::adjust
Percussion object
vptr
&Percussion::play
&Percussion::what
&Percussion::adjust
Stringed object
vptr
&Stringed::play
&Stringed::what
&Stringed::adjust
Brass object
vptr
&Brass::play
&Brass::what
&Wind::adjust

虚函数

```
class Pet {
public:
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
public:
  string speak() const { return "Bark!"; }
};

int main() {
  Dog ralph;
  Pet* p1 = &ralph;
  Pet& p2 = ralph;
  Pet p3;
  // Late binding for both:
  cout << "p1->speak() = " << p1->speak() <<endl;
  cout << "p2.speak() = " << p2.speak() << endl;
  // Early binding (probably):
  cout << "p3.speak() = " << p3.speak() << endl;
} ///:~
```



为什么需要虚函数

- 或者说，为什么不总是使用虚函数(像java那样)。
  - 效率是唯一的理由

抽象的基类和纯虚函数

- 在设计时，常常希望基类仅仅作为其派生类的一个接口，或者说，仅仅通过向上类型转换至基类来使用作为所有子类公共接口的方法。
- 基类不能，并且也不应该被实例化。
- 在一个抽象类中定义虚函数是一种有效的设计手段：
  - 完成了哪些功能，需要基类完成那些功能；
  - 通过一组公共接口来操纵一组类；

抽象的基类和纯虚函数

- 纯虚函数的语法：

```
class Instrument {
public:
  // Pure virtual functions:
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  // Assume this will modify the object:
  virtual void adjust(int) = 0;
};
```

抽象的基类和纯虚函数

- 纯虚函数
  - 不以实现为目的，使得类具有明显的抽象性；
  - 禁止对抽象类的函数以传值方式调用，避免了对象切片；

对象切片

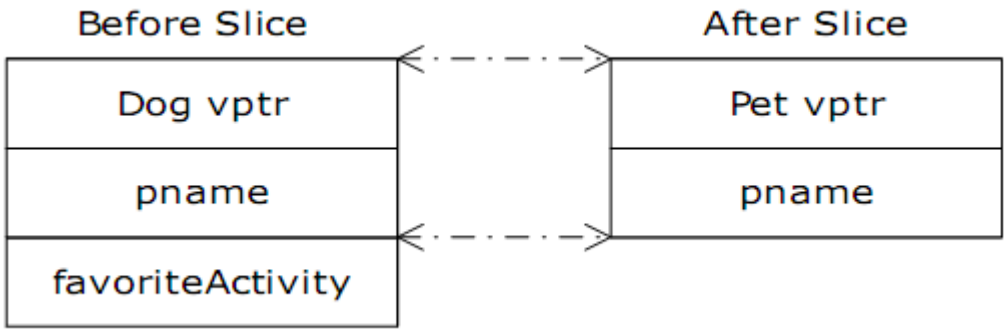
- 参数传递时，传地址和传值有很大的差别：
  - 地址长度是固定的，而值的长度根据对象的类型不同而不同。
- 考虑传值的情况，如果一个传值的参数的类型是一个有多个子类型的基类，传递参数时将发生什么情况？

对象切片

```
class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
      favoriteActivity;
  }
};

void describe(Pet p) { // Slices the object
  cout << p.description() << endl;
}
```

对象切片

```
int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
} ///:~
```



对象切片

- 问题
  - 带有纯虚函数的类是否能以传值的方式作为参数传递？

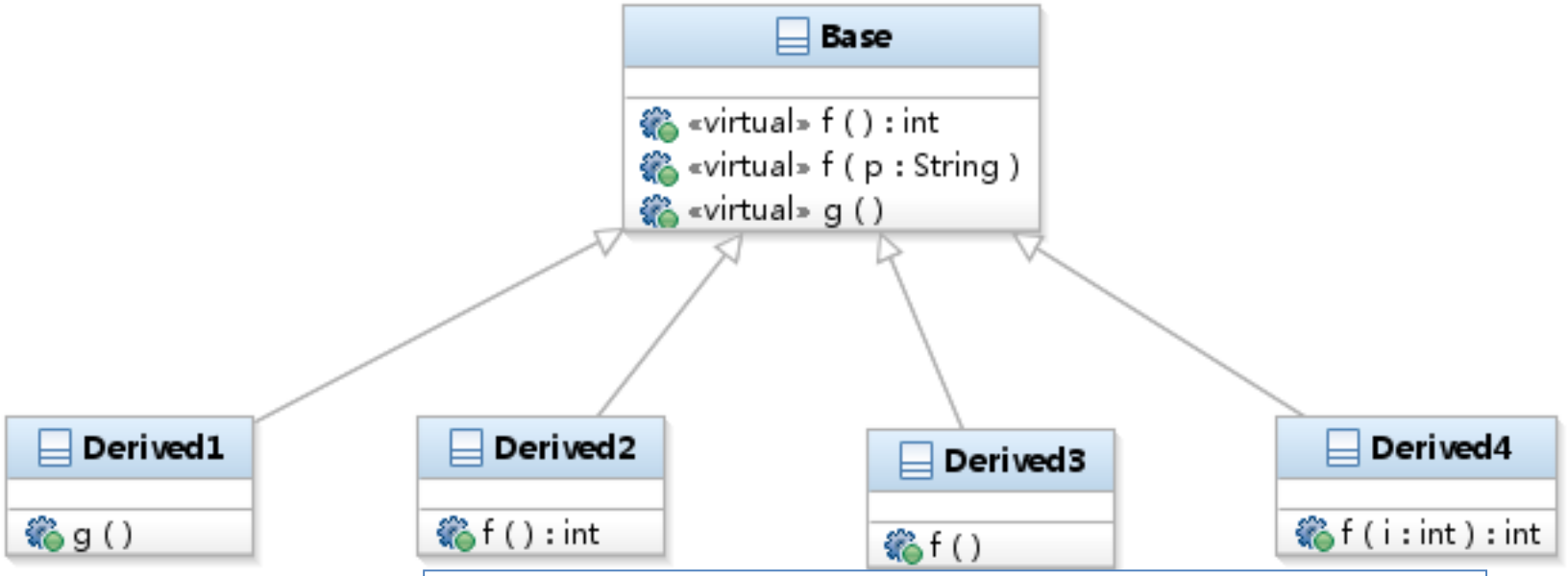
重载和覆盖(overload/override)

- overload 表示重载，用于同一个类中同名方法不同参数（包括类型不同或个数不同）的实现。
- override 表示覆盖，用于继承类对基类中虚成员的实现或重新定义。
  - 覆盖要求函数名/参数类型和个数完全一致

重载与覆盖(overload/override)

```
class Base {
public:
  virtual int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  virtual void f(string) const {}
  virtual void g() const {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Overriding a virtual function:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
```

```
class Derived3 : public Base {
public:
  // Cannot change return type:
  //! void f() const{ cout << "Derived3
};

class Derived4 : public Base {
public:
  // Change argument list:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};
```




```
int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version hidden
  Derived4 d4;
  x = d4.f(1);
//!  x = d4.f(); // f() version hidden
//!  d4.f(s); // string version hidden
  Base& br = d4; // Upcast
//!  br.f(1); // Derived version unavailable
  br.f(); // Base version available
  br.f(s); // Base version abailable
} ///:~
```

覆盖时变量的返回值

- 在覆盖一个返回基类的指针或引用的函数时，可以将返回值重定义为其基类的子类。

函数返回类型

```
class PetFood {
public:
  virtual string foodType() const = 0;
};

class Pet {
public:
  virtual string type() const = 0;
  virtual PetFood* eats() = 0;
};

class Bird : public Pet {
public:
  string type() const { return "Bird"; }
  class BirdFood : public PetFood {
  public:
    string foodType() const {
      return "Bird food";
    }
  };
  // Upcast to base type:
  PetFood* eats() { return &bf; }
private:
  BirdFood bf;
};
```

函数返回类型

```
class Cat : public Pet {
public:
  string type() const { return "Cat"; }
  class CatFood : public PetFood {
  public:
    string foodType() const { return "Birds"; }
  };
  // Return exact type instead:
  CatFood* eats() { return &cf; }
private:
  CatFood cf;
};
```

```
int main() {
  Bird b;
  Cat c;
  Pet* p[] = { &b, &c, };
  for(int i = 0; i < sizeof p / sizeof *p; i++)
    cout << p[i]->type() << " eats "
         << p[i]->eats()->foodType() << endl;
  // Can return the exact type:
  Cat::CatFood* cf = c.eats();
  Bird::BirdFood* bf;
  // Cannot return the exact type:
//!  bf = b.eats();
  // Must downcast:
  bf = dynamic_cast<Bird::BirdFood*>(b.eats());
} ///:~
```


```
int main() {
  Bird b;
  Cat c;
  Pet* p[] = { &b, &c, };
  for(int i = 0; i < sizeof p / sizeof *p; i++)
    cout << p[i]->type() << " eats "
         << p[i]->eats()->foodType() << endl;
  // Can return the exact type:
  Cat::CatFood* cf = c.eats();
  Bird::BirdFood* bf;
  // Cannot return the exact type:
//!  bf = b.eats();
  // Must downcast:
  bf = dynamic_cast<Bird::BirdFood*>(b.eats());
} ///:~
```

问题

- Java是否对返回值类型的处理与C++一样？
- Java是否有方法隐藏的概念？

虚函数和构造函数

- 构造函数中的隐含代码
  - 编译器将自动在构造函数中插入VPTR指针的初始化代码
  - 尽可能不要将构造函数声明为内联(inline)
- 构造函数的调用次序
  - 正如在上一章介绍的，基类的构造函数一定会被调用，并且是在子类的构造函数之前调用。

虚函数和构造函数

- 在构造函数中调用虚函数时：
  - 只是调用的该虚函数的本地版本
  - 虚函数的机制在构造函数中不起作用
  - 原因：包括指向虚函数表的指针在内的所有数据成员没有完成构造
- Never call virtual functions during construction or destruction

```
1  class base
2  {
3  public:
4      base()
5      {
6          func();
7      }
8
9      virtual void func()
10     {
11         puts("base");
12     }
13 };
14
15
16 class test : public base
17 {
18
19 public:
20     virtual void func()
21     {
22         puts("test");
23     }
24 };
25
26 int main()
27 {
28     test x;
29     return 0;
30 }
```

析构函数和虚析构函数

- 构造函数不能为虚函数，而析构函数常常必须是虚的。
  - 这是因为应该从实际类型的析构函数开始，向其基类方向层层执行析构的过程。
  - 一旦对象被构造，其VPTR就被初始化，所以可以发生虚函数调用
- 析构函数中调用虚函数
  - 同构造函数一样，析构函数中的虚函数调用机制也被禁止。

虚析构函数

- 虚析构函数的意义：基类的指针指向派生类对象，用基类的指针删除派生类对象

```
class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
  Base1* bp = new Derived1; // Upcast
  delete bp;
  Base2* b2p = new Derived2; // Upcast
```

纯虚析构函数

- 纯虚析构函数使得基类是抽象类（如果基类的任何其它函数是纯虚函数也一样）
- 目的：
  - 阻止基类的实例化

向下类型转换

- 与向上类型转换相对应的是向下类型转换：将一个类型转换为一种更特殊的类型。
- 什么是安全的类型转换？
  - 能够给出是否转换成功的信息的类型转换机制
- 使用dyanmic_cast关键字进行类型转换时，如果成功，则返回结果对象，否则返回0。
  - type-safe downcast
  - RTTI机制（run time type information），允许我们得到在进行向上类型转换时丢失的类型信息

向下类型转换

```
class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast
  // Try to cast it to Dog*:
  Dog* d1 = dynamic_cast<Dog*>(b);
  // Try to cast it to Cat*:
  Cat* d2 = dynamic_cast<Cat*>(b);
  cout << "d1 = " << (long)d1 << endl;
  cout << "d2 = " << (long)d2 << endl;
} ///:~
```

总结

- 多态的定义
  - virtual 关键字
- 多态的内部实现
  - VTABLE和VPTR
- 重载与覆盖(overload/override)
- 多态与构造函数/析构函数
- 多态与向下类型转换